Sign Determination in Residue Number Systems

Hervé Brönnimann* Ioannis Z. Emiris* Victor Y. Pan‡ Sylvain Pion*

June 14, 1999

Abstract

Sign determination is a fundamental problem in algebraic as well as geometric computing. It is the critical operation when using real algebraic numbers and exact geometric predicates. We propose an exact and efficient method that determines the sign of a multivariate polynomial expression with rational coefficients. Exactness is achieved by using modular computation. Although this usually requires some multiprecision computation, our novel techniques of *recursive relaxation of the moduli* and their variants enable us to carry out sign determination and comparisons by using only single precision. Moreover, to exploit modern day hardware, we exclusively rely on floating point arithmetic, which leads us to a hybrid symbolic-numeric approach to exact arithmetic. We show how our method can be used to generate robust and efficient implementations of real algebraic and geometric algorithms including Sturm sequences, algebraic representation of points and curves, convex hull and Voronoi diagram computations and solid modeling. This method is highly parallelizable, easy to implement, and compares favorably with known multiprecision methods from a practical complexity point of view. We substantiate these claims by experimental results and comparisons to other existing approaches.

1 Introduction

Manipulation of real numbers on modern computers is largely done by using fixed-precision floating-point arithmetic (f.p. arithmetic for short). Consequently, and due to the importance of numerical computing, f.p. arithmetic has benefited from important infrastructural support, and extremely efficient hardware implementations are available. F.p. arithmetic is only approximate, however. While this may be acceptable in performing numerically stable computations, it introduces many limitations and is unacceptable in algebraic computation and in deciding geometric predicates. The goal of this paper is to use fixed-precision f.p. arithmetic for performing exact computations, in order to decrease their complexity. A major application of our algorithms is in manipulating real algebraic numbers exactly, thus offering an alternative for an exact computation over the reals.

In computer algebra and symbolic computation, exact arithmetic is almost always assumed. When approximate calculation is not an option, a popular approach is to use big-integer and big-float multiprecision packages. This implies that operands are computed and stored with arbitrary precision, including intermediate quantities whose magnitude may be significantly larger than that of the output values. To remedy this problem, a substantial amount of work in the area has focused on modular arithmetic, which allows most of the computation to be carried over fixed precision integers. However, the modular representation of a rational number is typically not sufficient, and most problems require the reconstruction of the exact number, which means that some arbitrary precision is still required. The main contribution of

*INRIA Sophia-Antipolis, B.P. 93, 2004, Route des Lucioles, 06902 Sophia-Antipolis Cedex, FRANCE. This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

‡ Department of Mathematics and Computer Science, Lehman College, City University of New-York, Bronx NY 10468, USA. Supported by NSF Grant CCR 9625344 and PSC CUNY Award No. 667340.

this paper are algorithms that determine the sign of such a number by using single precision operations on its moduli, thus removing the need of any high precision computation. Real algebraic numbers are represented as the unique root of a given polynomial in a given interval. Such a representation can be computed by applying Sturm theory. Besides the computation of Sturm sequences, finding the isolating interval requires many computations of signs of polynomial expressions with integer coefficients. The major drawback of these methods is the slowdown due to the handling of full precision, so our algorithms are ideally suited to this problem and analyzed in section 6.1.

In computational geometry, computer-aided design (CAD), geometric modeling and computer graphics, on the other hand, f.p. arithmetic is extremely popular because of its speed. Most of geometric predicates can be expressed as computing the sign of an algebraic expression, which can be computed by using f.p. arithmetic with a fixed finite precision. Unfortunately, the roundoff errors may easily lead to the wrong sign, causing the algorithm to fail on the input. This problem is often referred to as the *robustness* problem [24]. One solution to the robustness problem is to explicitly handle numerical inaccuracies, so as to design an algorithm that does not fail even if the numerical part of the computation is done approximately [25, 37], or to analyze the error due to the f.p. imprecision [19]. Such designs are extremely involved and have only been done for a few algorithms. The general solution, it has been widely argued, is to compute certain predicates exactly [11, 15, 17, 20, 41] (see also section 6.2). This is also the position taken by this paper. This goal can be achieved in many ways: computing the algebraic expressions with infinite precision [39], with a finite but much higher precision that can be shown to suffice [21], or by using an algorithm that performs a specific test exactly. In the last category, much work has focused on computing the sign of the determinant of a matrix with integer entries [3, 10, 13], which applies to many geometric tests (such as orientation tests, in-circle tests, comparing segment intersections) as well as to algebraic primitives (such as resultants and algebraic representations of curves and surfaces). Recently, some techniques have been devised for handling arbitrary polynomial expressions and f.p. representation [36] but their complexity grows rather fast with that of the computation.

This paper proposes a method that determines exactly the sign of a multivariate polynomial with rational coefficients evaluated at a rational point. It uses no operations other than modular arithmetic and f.p. computations with a fixed finite (single) precision. We call the new technique the *recursive relaxation of the moduli*. The key feature is the exact result, combined with the efficiency of f.p. arithmetic on modern day computers. Our methods can be used in many settings, including the operations in computer algebra and exact geometric predicates mentioned above, as well as whenever numerical techniques need an exact test; see section 6 for examples. In short, our methods combine the veracity of exact computation with the speed of f.p. arithmetic and, therefore, contribute to the current initiative of cross-fertilizing the areas of numeric and symbolic computing. Moreover, we propose several variants of our techniques improving some aspects of the algorithm.

More specifically, our algorithms perform rational algebraic computations modulo several primes, that is, with a lower precision. As shown in section 2, this requires only single-precision f.p. operations. The Chinese remainder theorem enables us to combine the resulting values together in order to recover the desired output value. This is not a new trick: such a representation of integers by their moduli is known as *Residue Number Systems* (RNS) and is popular because its provides a cheap and highly parallelizable version of multiprecision arithmetic. It is impossible here to give a fair and full account on RNS, but [1, 28, 29, 35] provide a good introduction to the topic. From a complexity point of view, RNS allow to add and multiply numbers in linear time. The latter stage of combining the moduli to reconstruct the explicit answer, however, was always perceived as the bottleneck of this approach because higher precision computations were required [35, chap. 4] The recursive relaxation of the moduli enables us to greatly accelerate this phase, since it only needs some simple single precision computations.

The closest predecessors of our work are apparently [16], [26], and [7]. The algorithm of Hung and Parhami [26] corresponds to single application of the second stage of our recursive relaxation of the moduli. Such a single application suffices in the context of the goal of [26], that is, application to divisions in RNS, but in terms of the sign determination of an integer, this only works for an absolutely larger input. For smaller inputs, Bajard, Didier and Müller [7] keep a floating point estimate of the number, which allows to guess beforehand how many loops of our generalized Lagrange algorithm should be executed. We note that their technique handles overflows gracefully, but it cannot handle integers larger than those stored in the floating point representation, which may be quite limiting. The maximum range

of exponents in the IEEE 754 Standard is 2024. Our moduli are on the order of 2^{27}. Therefore, Bajard, Didier and Müller allow integers representable on 74 moduli only. In contrast, our technique allows for more than 10^7 moduli. The article [16] gives probabilistic estimates for early termination of Newton's interpolation process, which we apply in our probabilistic analysis of our algorithm 5. Its main subject is an implementation of an algorithm computing multidimensional convex hulls. The article [16] does not use our techniques of recursive relaxation of the moduli and does not discuss Lagrange's approach. The spirit of the present article is shared by [18], where f.p. computation is used to calculate the most significant part of the answer, whereas modular arithmetic yields the least significant part. A limited subset of our results in preliminary form appeared as [8].

Here is the outline of the paper. The next section introduces the different arithmetics and defines the problem at hand. Sections 3 and 4 correspond to the two algorithms for determining the sign of an arbitrary rational defined by a set of moduli by performing single precision f.p. operations. Section 5 elaborates some variants of these algorithms. Sections 3-5 constitute our main contribution and propose deterministic as well as probabilistic algorithms. They are applied to computing over real algebraic numbers, to determining exact geometric predicates, and to the ubiquitous question of determinant sign, in section 6. The experimental results of section 7 support our claim that our algorithms are the fastest today in practice. Our main results are summarized in section 8.

2 Exact sign computation using modular arithmetic

Floating point (f.p.) computations. Our model of a computer is that of a f.p. processor that performs operations at unit cost by using b-bit precision (e.g., in the IEEE 754 double precision standard, we have $b = 53$). It is a realistic model as it covers the case of most workstations used in research and industry [22, 28, 36]. We will use mainly one basic property of f.p. arithmetic on such a computer: for all four arithmetic operations, the computed result is always the f.p. representation that best approximates the exact result. This means that the relative error incurred by an operation returning x is at most 2^{-b-1}, and that the absolute error[1] is at most $2^{\lfloor \log |x| - b - 1 \rfloor}$. In particular, operations performed on pairs of integers smaller than 2^b are performed exactly as long as the result is also smaller than 2^b.

To be able to discuss the properties of f.p. arithmetic, it is convenient to introduce the following notation [36]: given any real number x, it is *representable*[2] over b bits if $x = 0$ or if $x2^{-\lfloor \log x \rfloor + b}$ is an integer; $\widetilde{x}$ denotes the representable f.p. number closest to x (with any tie-breaking rule if x is right in-between two representable numbers), and $\mathrm{ulp}(x)$ denotes the *unit in the last place*, that is, $2^{\lfloor \log |x| - b \rfloor}$ if $x \neq 0$, and 0 otherwise. With this notation, the absolute error in computing an operation that returns x is $\frac{1}{2}\mathrm{ulp}(x)$.

Modular computations. Let $m_1, \ldots, m_k$ be k pairwise relatively prime integers and let $m = \prod_i m_i$. For any number x (not necessarily an integer), we let $x_i = x \bmod m_i$ be the only number in the range $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ such that $x_i - x$ is a multiple of m_i. (This operation is always among the standard operations because it is needed for reducing the arguments of periodic functions.)

This operation can be extended modulo an f.p. numbers as follows: an f.p. number x is truncated to a non-null f.p. number y and the result is defined as $x - \lceil x/y \rfloor y$. Therefore, $x \bmod m_i$ is the result of truncating x to m_i, and the (signed) fractional part $\mathrm{frac}(x)$ of x is the result of truncating x to 1. Note that the result of truncating x to a power of two is always representable if x is representable.

To be able to perform arithmetic modulo m_i on integers by using f.p. arithmetic with b-bit precision, we will assume that $m_i \leq 2^{b/2+1}$. Performing modular multiplications of two integers from the interval $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ can be done by multiplying these numbers and returning their product modulo m_i. (The product is smaller than 2^b in magnitude and hence is computed exactly.) Performing additions can be done very much in the same way, but since the result is in the range $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, taking the sum modulo m_i can be achieved by adding or subtracting m_i if necessary. Modular divisions can be computed using

[1] All logarithms in this paper are base 2.

[2] We systematically ignore underflows and overflows, by assuming that the range of exponent is large enough. A few modern packages now provide f.p. arithmetic with the exponent stored in a separate integers, which extends the IEEE 754 double precision standard by quite a lot.

the extended Euclid algorithm; we will need them in this paper only in section 6. Therefore, arithmetic modulo m_i can be performed on integers by using f.p. arithmetic with b-bit precision, provided that $m_i \leq 2^{b/2+1}$.

Exact sign computation. In this paper, we consider the following computational problem.

Problem 1 *Let k, b, $m_1, \ldots, m_k$ denote positive integers, $m_1, \ldots, m_k$ being pairwise relatively prime, such that $m_i \leq 2^{b/2+1}$, and let $m = \prod_{i=1}^{k} m_i$. Let x be an integer whose magnitude is smaller than $\lfloor m/2 \rfloor$. Given $x_i = x \bmod m_i$, compute the sign of x by using only modular and floating-point arithmetic both performed with b-bit precision.*

We will solve this problem, even though x can be huge and, therefore, not even representable by using b bits. In the worst case, our solutions require $O(k^2)$ operations and therefore do not improve asymptotically over the standard multiprecision approach. They are simple, however, and require little or no overhead. In practice, they only perform $O(k)$ operations. Thus they are very well suited for implementation.

3 Lagrange's method

According to the Chinese remainder theorem, x is uniquely determined by its residues x_i, that is, Problem 1 is well defined and admits a unique solution. Moreover, this solution can be derived algorithmically from a formula due to Lagrange. A comprehensive account of this approach can be found in [28, 29].

3.1 The basic method

This section describes the basic algorithm relying on Lagrange's approach. If x is an integer in the range $\left[-\frac{m}{2}, \frac{m}{2}\right)$, x_i stands for the residue $x \bmod m_i$, $v_i = m/m_i = \prod_{j \neq i} m_j$, and $w_i = v_i^{-1} \bmod m_i$, then

$$x = \left(\sum_{i=1}^{k} ((x_i w_i) \bmod m_i)\, v_i \right) \bmod m. \tag{1}$$

Trying to determine the sign of such an integer, we compute the latter sum approximately in fixed b-bit precision. Computing a linear combination of large integers v_i with its subsequent reduction modulo m can be difficult, so we prefer to compute the number

$$S = \frac{x}{m} = \operatorname{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i) \bmod m_i}{m_i} \right),$$

where $\operatorname{frac}(z)$ is the fractional part of a number x that belongs to $\left[-\frac{1}{2}, \frac{1}{2}\right)$.

If S were computed exactly, then we would have $S = x/m$, due to Lagrange's interpolation formula. In fact, S is computed with a fixed b-bit precision. Nevertheless, if we compute it by incrementally adding the ith term and taking fractional part, the error bound follows the induction

$$\varepsilon_i = \varepsilon_{i-1} + 2^{-b-1} + 2^{-b},$$

where the term 2^{-b-1} accounts for the error on computing the ith term of S, and the term 2^{-b} accounts for the error on computing the incremental sum. Moreover, $\varepsilon_1 = 2^{-b-1}$. A technical problem can arise if S is too close to a half-integer, because the fractional part may not be computed properly. We circumvent this by assuming that $|x|$ is less than $\frac{m}{2}(1 - \varepsilon_k)$. In this way, we can insure that S approximates x/m within an absolute error bound $\varepsilon_k = (3k - 2)2^{-b-1}$.

Therefore, if $|S|$ is greater than ε_k, the sign of x is the same as the sign of S, and we are done. Otherwise, $|x| \leq 2\varepsilon_k m$. Since $m_k \leq 2^{b/2+1}$, we can say conservatively that $2\varepsilon_k m$ is smaller than $\frac{m}{2m_k}(1 - \varepsilon_{k-1})$, for all practical values of k and b, and hence we may recover x already from $x_i = x \bmod m_i$ for $i = 1, \ldots, k-1$, that is, it suffices to repeat the computation using only $k - 1$, rather than k moduli.

Recursively, we will reduce the solution to the case of a single modulus m_1 where $x = x_1$. We will call this technique *recursive relaxation of the moduli*, and we will also apply it in section 3.2.

We will present our resulting algorithm by using additional notation:

$$\begin{aligned} m^{(j)} &= \prod_{1 \le i \le j} m_i, \\ v_i^{(j)} &= \prod_{\substack{1 \le \ell \le j \\ \ell \ne i}} m_\ell, \\ w_i^{(j)} &= \left(v_i^{(j)}\right)^{-1} \bmod m_i, \qquad (2) \\ S^{(j)} &= \operatorname{frac}\left(\sum_{i=1}^{j} \frac{x_i w_i^{(j)} \bmod m_i}{m_i}\right), \qquad (3) \end{aligned}$$

so that $m = m^{(k)}$, $v_i = v_i^{(k)}$, $w_i = w_i^{(k)}$ and $S = S^{(k)}$. All the computations in this algorithm are performed by using f.p. arithmetic with b-bit precision. Because this applies also to fractional part computation, we must assume that $x/m^{(k)}$ is sufficiently far from half-integers, hence we assume that

$$|x| \le \frac{m^{(k)}}{2}(1 - \varepsilon_k).$$

This assumption is not too restrictive since it is violated with very low probability ε_k for random x_i, and it can be remedied by computing one more residue x_{k+1}.

Algorithm 1 : Compute the sign of x knowing $x_i = x \bmod m_i$

Precomputed data: m_j, $w_i^{(j)}$, ε_j, *for all* $1 \le i \le j \le k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, *for all* $1 \le i \le k$
Output: *sign of* x, *the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$
Precondition: $|x| \le \frac{m^{(k)}}{2}(1 - \varepsilon_k)$

1. *Let* $j \leftarrow k + 1$
2. *Repeat* $j \leftarrow j - 1$,
   $S^{(j)} \leftarrow frac\left(\sum_{i=1}^{j} \frac{x_i w_i^{(j)} \bmod m_i}{m_i}\right)$
   until $|S^{(j)}| > \varepsilon_j$ *or* $j = 0$
3. *Return sign of* $S^{(j)}$

The following lemma bounds the number of operations performed by the algorithm in the worst case.

Lemma 3.1 *Algorithm 1 computes the sign of x knowing its residues x_i by using at most $\frac{k(k-1)}{2}$ modular multiplications, $\frac{k(k-1)}{2}$ f.p. divisions, $\frac{k(k-1)}{2}$ f.p. additions, and $k + 2$ f.p. comparisons. All of these operations can be implemented in f.p. arithmetic.*

Proof. From the preceding discussion, it is clear that the algorithm reports the sign of x. Indeed, upon termination we have $\varepsilon_j m^{(j-1)} < |x| \le \frac{m^{(j)}}{2}(1 - \varepsilon_j)$, or else $j = 0$. In any case, $|S^{(j)}|$ is an approximation of $|x|/m^{(j)}$ with a sufficiently small relative error (bounded by ε_j) so that the sign of $S^{(j)}$ is exactly the sign of x.

The m_i's and the $w_i^{(j)}$'s are computed once and for all and placed into a table, so they are assumed to be available to the algorithm at unit cost. In step 2, a total of j modular multiplications, j f.p. divisions, and j f.p. additions (including taking the fractional part) are performed. □

In almost all practical instances of the problem, $|x|$ is on the same order of magnitude as $m^{(k)}$. If $|x|$ is not too small compared to $m^{(k)}$, then only step k is performed, involving only at most k f.p. operations of each kind. This is to be contrasted with full reconstruction, which requires $\Theta(k^2)$ operations. Thus algorithm 1 is of great practical value.

Let us also formalize the latter argument. Since x is chosen independently with $m_1, \ldots, m_k$ being fixed, we may assume it corresponds to a random integer in the range $[-\frac{m}{2}(1-\varepsilon_k), \frac{m}{2}(1-\varepsilon_k)]$. Under the uniform probability distribution in this range, the probability that $|x| > 2\varepsilon_k m$ is $1 - 4\varepsilon_k/(1-\varepsilon_k)$. Since ε_k is extremely small, this probability is very close to 1 and, therefore, the algorithm most likely computes the sign of x correctly already at the first stage. The exception is the cases where $|x|$ is biased to be small as, for instance, when x represents the determinant of a nearly singular matrix. Such cases are well handled by the algorithms of section 5.

By using parallel implementation of the summation of k numbers on $\lceil k/\log k \rceil$ arithmetic processors in $2\lceil \log k \rceil$ time (cf. e.g. [5, ch.4]), we may perform algorithm 1 on $\lceil k/\log k \rceil$ arithmetic processors in $O(k \log k)$ time, assuming each b-bit f.p. operation takes constant time. Furthermore, if $\lceil k^2/\log k \rceil$ processors are available, we may compute all the $S^{(j)}$ and compare $|S^{(j)}|$ with ε_j, for all $j = 1, \ldots, k$ concurrently. This would require $O(\log k)$ time on $\lceil k^2/\log k \rceil$ processors. Finally, if $\lceil tk/\log k \rceil$ processors are available for some parameter $1 \leq t \leq k$, we may perform algorithm 1 in $O((k \log k)/t)$ time by batching $\lceil t \rceil$ consecutive values of j in parallel. In practice, the algorithm terminates well before j reaches k, so $O(\log k)$ time suffices even with $\lceil k/\log k \rceil$ processors.

Remark 1. If actually $x = 0$, the algorithm can be greatly sped up by testing if $x_j = 0$ in step 2, in which case we may directly pass to $j - 1$. Furthermore, stage 3 is not needed unless $x = x_j = 0$ for all j, which can be tested beforehand. Of course, if the only answer needed is "$x = 0$" or "$x \neq 0$", then it suffices to test if all the x_i's are zero.

Remark 2. The costly part of the computation is likely to be the determination of the x_i's. For these reasons, we should try to minimize the number k of moduli m_i involved in the algorithm. This can be done by getting better upper estimates on the magnitude of the output or by using the probabilistic technique of section 5.

3.2 A generalization of Lagrange's method

We will show that Lagrange's method is in fact a particular case of the following method [7]. Let

$$\Sigma^{(0)} = S^{(k)} = \text{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i) \bmod m_i}{m_i}\right).$$

This quantity is computed in the first step of algorithm 1. If the computed value of $\Sigma^{(0)}$ is smaller than ε_k, it implies that $\Sigma^{(0)} < 2\varepsilon_k$. Thus, $|x|$ is smaller than $2m\varepsilon_k$. We can then multiply $x_i w_i$ by

$$\alpha_k = \left\lfloor \frac{\frac{1}{2}(1-\varepsilon_k)}{2\varepsilon_k} \right\rfloor,$$

to obtain $(x_i w_i \alpha_k) \bmod m_i$ for all $i = 1, \ldots, k$. This can easily be done by precomputing α_k modulo each m_i. We then compute

$$\Sigma^{(1)} = \text{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i \alpha_k) \bmod m_i}{m_i}\right),$$

and more generally,

$$\Sigma^{(j)} = \text{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i {\alpha_k}^j) \bmod m_i}{m_i}\right),$$

where we assume $\alpha_k \bmod m_i$ precomputed for all $i = 1, \ldots, k$. This leads to the following algorithm:

Algorithm 2 : Generalized Lagrange's method. Compute the sign of x knowing $x_i = x \bmod m_i$.

Precomputed data: m_i, w_i, ε_k, $\alpha_k \bmod m_i$, *for all* $i = 1, \ldots, k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for all* $i = 1, \ldots, k$
Output: *sign of* x, *the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m}{2}, \frac{m}{2}\right)$

Preconditions: $|x| \leq \frac{m}{2}(1-\varepsilon_k)$ *and* $x \neq 0$

1. *Let* $j \leftarrow -1$
2. *Repeat* $j \leftarrow j+1$,
   $\Sigma^{(j)} \leftarrow frac\left(\sum_{i=1}^{k} \frac{x_i w_i \bmod m_i}{m_i}\right)$
   if $|\Sigma^{(j)}| \leq \varepsilon_k$ *and* $j < k$ *then* $x_i \leftarrow x_i \alpha_k \bmod m_i$ *for all* $1 \leq i \leq qk$,
   until $|\Sigma^{(j)}| > \varepsilon_k$ *or* $j = k$
3. *If* $j = k$ *return* "$x = 0$"
4. *If* $\Sigma^{(j)} > 0$ *return* "$x > 0$"
5. *If* $\Sigma^{(j)} < 0$ *return* "$x < 0$"

It is easy to see that the number of iterations in step 2 is $O(\log m / \log \alpha_k) = O(k)$, because $|x|$ is no less than 1 and no more than $m^{(k)} \leq 2^{k(b/2+1)}$, and is multiplied by α_k at each iteration. Therefore, algorithm 2 still performs $\Theta(k^2)$ operations in the worst case, but in practice (on most instances) only k operations of each kind

Remark 3. Algorithm 1 corresponds to a choice of m_{k-j} instead of α_k in step j. This simplifies the computation by eliminating one modulus at each iteration, but it performs more iterations. Multiplying by α_k, we perform about twice fewer iterations (α_k is on the order of two moduli) but each iteration is done with k moduli. This is why we call algorithm 2 a generalization.

It is possible to combine the techniques of both algorithms by replacing α_k in the expression for $\Sigma^{(j)}$ by the greatest $\beta_{k,j} \leq \alpha_k$ which is a multiple of $m^{(j)}$. This means that $\beta_{k,j}$ is also precomputed.

Remark 4. To yield the parallel time bounds such as $O(\log k)$ using $\lceil k^2/\log k \rceil$ processors for algorithm 2, we need to precompute ${\alpha_k}^j \bmod m_i$ for all $i, j = 1, \ldots, k$.

4 Newton's method

An incremental version of Chinese remainder reconstruction, named after Newton, is described in this section. We recall the method for completeness; see [28, 29] for a comprehensive presentation. Its main advantage is that it can be adapted to a probabilistic algorithm that does not require an *a priori* bound on the magnitude of x. This is the subject of section 5.3.

Let $x^{(j)} = x \bmod m^{(j)}$, for $j = 1, \ldots, k$, so that $x^{(1)} = x_1$ and $x = x^{(k)}$. Let $y_1 = x_1$, and for all $j = 2, \ldots, k$,

$$\begin{aligned} t_j &= w_j^{(j)} = (m^{(j-1)})^{-1} \bmod m_j, \\ y_j &= \left(x_j - x^{(j-1)}\right) t_j \bmod m_j \quad \in \left[-\frac{m_j}{2}, \frac{m_j}{2}\right). \end{aligned}$$

Then, for all $j = 2, \ldots, k$,

$$x^{(j)} = \left(x^{(j-1)} + y_j m^{(j-1)}\right) \bmod m^{(j)}. \tag{4}$$

Clearly, this leads to an incremental computation of the solution $x = x^{(k)}$ to problem 1. An advantage is that all computation can be kept modulo m_j, and no floating-point computation is required, in contrast to sections 3.1 and 3.2 where $S^{(j)}$ or $\Sigma^{(j)}$ are computed. The y_j define the mixed-radix representation of x, which would offer an alternative way to perform arithmetic on long integers; see also [28, 35]. It is obvious, that when $y_j \neq 0$, then the sign of $x^{(j)}$ is the same as the sign of y_j since $|x^{(j-1)}| \leq m^{(j-1)}/2$. If $y_j = 0$, the sign of $x^{(j)}$ is the same as that of $x^{(j-1)}$, for $j \geq 2$, whereas the sign of $x^{(1)} = x_1 = y_1$ is known. If $y_j = 0$ for all j, then this is precisely the case when $x = 0$.

For $1 \leq i < j \leq k$, we introduce integers

$$u_i^{(j-1)} = \left(m^{(i-1)} t_j\right) \bmod m_j = \left(\prod_{l=i}^{j-1} m_l\right)^{-1} \bmod m_j.$$

Then $t_j = u_1^{(j-1)}$. Unrolling equation (4) in the definition of y_j shows that the quantities y_j verify the following equality for all $j = 2, \ldots, k$:

$$y_j = \left((x_j - x_1) u_1^{(j-1)} - \sum_{i=2}^{j-1} y_i u_i^{(j-1)} \right) \bmod m_j .$$

Therefore, they can be computed by using modular arithmetic with bit-precision given by the maximum bit-size of the m_j^2. Here it suffices to assume that the absolute value of x is bounded by $m^{(k)}/2$.

Algorithm 3 : Compute the sign of x, knowing $x \bmod m_i$, by Newton's incremental method.

Precomputed data: m_j, $u_i^{(j-1)}$, *for all* $1 \le i < j \le k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for all* $i = 1, \ldots, k$
Output: *sign of* x, *where* x *is the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$
Precondition: *None.*

1. *Let* $y_1 \leftarrow x_1$, $j \leftarrow 1$. *Set* s *to* $-1, 0$ *or* 1, *if* y_1 *is negative, zero or positive, respectively.*
2. *Repeat* $j \leftarrow j + 1$,
   $$y_j \leftarrow \left((x_j - x_1) u_1^{(j-1)} - \sum_{i=2}^{j-1} y_i u_i^{(j-1)} \right) \bmod m_j ,$$
   Set s *to* 1 *or* -1, *if* y_j *is positive or negative, respectively.*
   If $y_j = 0$ *then* s *does not change.*
   until $j = k$.
3. *Return sign of* s

Remark 5. As in remark 1, we can test beforehand if all $x_i = 0$, which is precisely the case when $x = 0$.

Lemma 4.1 *Algorithm 3 computes the sign of* x *knowing its residues* x_i *using exactly* $\frac{k(k-1)}{2}$ *modular multiplications,* $\frac{k(k-1)}{2}$ *modular additions, and* $2k$ *comparisons. All of these operations can be implemented in f.p. arithmetic.*

Proof. For every $j = 2, \ldots, k$, there are $j - 1$ modular additions and multiplications. There is one sign computation for each $j = 1, \ldots, k$, each of which can be implemented by two comparisons. □

Algorithm 3 requires k recursive steps, so its parallel time cannot be decreased below $\Omega(k \log k)$. Nevertheless the algorithm can be implemented in $O(k \log k)$ time on $\lceil k / \log k \rceil$ processors, assuming each b-bit f.p. operation takes constant time.

To compare with algorithm 1, realistically assume that a modular addition is equivalent to 3/2 f.p. additions and one comparison, on the average. Then, algorithm 1 requires at most $\frac{k(k-1)}{2}$ f.p. divisions (which are essentially multiplications with precomputed reciprocals) more than algorithm 3, whereas the latter always requires $\frac{k(k-1)}{4}$ extra f.p. additions and $\frac{k(k+1)}{2}$ additional comparisons.

5 Variants

Both methods require precomputed tables whose sizes are quadratic. Using a Horner-like scheme, it is simple to reduce the sizes of these tables to linear, as we show in the next subsection.

Also, both methods can be adapted to yield probabilistic algorithms. Such an algorithm reports a wrong answer with a (very small) probability. The advantage of the first algorithm is that it parallelizes the sign detection method in polylogarithmic time. The advantage of the second algorithm is that it may require to compute much fewer moduli.

5.1 Reducing the size of precomputed tables

In algorithms 1, 2, and 3, the size of the precomputed tables is quadratic in k. This can clearly be a limitation for large values of k, for which our algorithms are of highest interest. Below, we indicate how to reduce the size of the precomputed tables to linear for fixed k for algorithms 1, 2, and to linear for variable k for algorithm 3.

Assume that k is fixed in the use of the algorithm 1. Under the notation of section 3, we notice that

$$w_i^{(j-1)} = w_i^{(j)} m_j \bmod m_i.$$

Therefore, if $z_i = x_i w_i^{(j)} \bmod m_i$ in step 2 of algorithm 1, we see that at the next iteration of step 2 (before $j \leftarrow j-1$), $z_i \leftarrow z_i \times m_j \bmod m_i$ will update the value of z_i correctly. We thus modify the algorithm as follows:

Algorithm 4 : Lagrange's method with table of linear size only for fixed k

Precomputed data: m_j, $w_i^{(k)}$, ε_j, *for all* $1 \le j \le k$ *and all* $1 \le i \le k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, *for all* $1 \le i \le k$
Output: *sign of* x, *the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$
Precondition: $|x| \le \frac{m^{(k)}}{2}(1-\varepsilon_k)$

1. *Let* $j \leftarrow k$, $z_i = x_i w_i^{(k)} \bmod m_i$ *for all* $1 \le i \le k$,
   $S^{(k)} \leftarrow frac\left(\sum_{i=1}^{k} \frac{z_i}{m_i}\right)$
2. *While* $|S^{(j)}| \le \varepsilon_j$ *and* $j > 0$, *do*
   $z_i = z_i m_j \bmod m_i$ *for* $1 \le i < j$
   $j \leftarrow j-1$
   $S^{(j)} \leftarrow frac\left(\sum_{i=1}^{j} \frac{z_i}{m_i}\right)$
3. *Return sign of* $S^{(j)} > 0$.

For algorithm 3, we notice as indicated in [28, Ex.4.3.2(7)] that $u_{i+1}^{(j)} = u_i^{(j)} m_i$. Since $u_1^{(j)} = t_j$, this suggests the use of the following Horner-like scheme:

$$y_j = t_j\left(x_j - x_1 - m_1\left(y_2 - m_2\left(y_3 - \cdots m_{j-2} y_j - 1\right)\cdots\right)\right) \bmod m_j.$$

Replacing this expression in step 2 of algorithm 3 shows that only the precomputed quantity t_j needs to be computed for all j in the desired range. Thus the precomputed table for algorithm 3 modified is only of size linear in the maximum number of moduli.

5.2 A probabilistic variant of Lagrange's method

This section introduces a probabilistic approach based on Lagrange's method. In algorithm 2, there can be at most $h_{worst} = \left\lceil \log(2m^{(k)}\varepsilon_k - 1)/\log \alpha_k \right\rceil$ iterations. The actual number h_{actual} of iterations is the minimum h that satisfies $|x{\alpha_k}^h| \ge 2m^{(k)}\varepsilon_k$. In that algorithm, we find this number by repeatedly incrementing h. In theory we could perform a binary search on h by testing whether $|x{\alpha_k}^h| \ge 2m^{(k)}\varepsilon_k$. Since the value of x is unknown, however, we can only test if $|x{\alpha_k}^h \bmod m^{(k)}| \ge 2m^{(k)}\varepsilon_k$ by using step 2 of the algorithm. If this is detected to hold for some value of h, then necessarily $|x{\alpha_k}^h| \ge 2m^{(k)}\varepsilon_k$, i.e. $h_{actual} \le h$, and we should try a smaller value of h. Otherwise, it is only a probabilistic indication that $|x{\alpha_k}^h| < 2m^{(k)}\varepsilon_k$, i.e. $h_{actual} > h$, but we may try nevertheless a greater value of h.

We therefore begin with h, and then double the value of h until the condition $|x{\alpha_k}^h \bmod m^{(k)}| \ge 2m^{(k)}\varepsilon_k$ is true. Then the range $[0, h]$ is guaranteed to contain the value h_{actual}. We may then perform a binary search for h_{actual} in the range $[0, h]$. Then for any intervals $[h_-, h_+)$ computed in the binary search, h_+ is a guaranteed upper bound, whereas h_- is a lower bound only with a high probability.

When the interval reduces to $[h, h+1)$, the sign can be determined by using algorithm 2. We call this technique *binary search in a randomized range*.

Since $2\varepsilon_k$ is much smaller than 1, the probability that, for some fixed h, k, α_k, a random x in the range $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$ satisfies $|x{\alpha_k}^h \bmod m^{(k)}| \geq 2m^{(k)}\varepsilon_k$ but not $|x{\alpha_k}^h| \geq 2m^{(k)}\varepsilon_k$ is extremely small, yet nonzero. The speedup is obtained by the fact that only $O(\log k)$ iterations are processed. The resulting algorithm performs only $O(k \log k)$ operations. It may be executed on $\lceil k/\log k \rceil$ processors in parallel time $O(\log^2 k)$.

Remark 6. It is possible — although we do not detail it here — to remove the probabilistic aspect of this technique, that is, to certify whether the lower bound h_- is correct. This method may require, in the worst case, quadratic time but, with high probability, it has time complexity in $O(n \log n)$, just like our probabilistic algorithm.

5.3 A probabilistic variant of Newton's method

We propose below a probabilistic variant of algorithm 3 which, moreover, removes the need of an *a priori* knowledge of k. The principal feature of Newton's approach is its incremental nature. In our variant, this may lead to faster termination, before examining all k moduli. Informally, this should happen whenever the magnitude of x is significantly smaller than $m^{(k)}/2$, in which case we would save the computation required to obtain x_j for all larger j. This saves a significant amount of computation if termination occurs earlier than the static bound indicated by k.

This occurs when the method is used in conjunction with some filter that handles the cases of large absolute value, or when the problem is such that the distribution of x is not uniform but is instead biased towards smaller values. An example is the construction of the convex hull facet structure, where the gain due to the probabilistic termination is quantified [16]. For example, the 7-dimensional convex hull of 100 points with integer coordinates of 31 bits is accelerated by more than 5%.

Step 2 is modified to include a test of y_j against zero. Clearly, $y_j = 0$ precisely when $x^{(j)} = x^{(j-1)}$. Then we may deduce that $x^{(j)} = x^{(k)} = x$, with a very high probability, and terminate the iteration. In terms of mixed-radix representation, this assumes that when $y_j = 0$ then all more significant y_i's will also be zero. This is no different from escaping in multiprecision arithmetic when some digit (or sequence of consecutive digits) turns out to be zero, assuming then that the higher order digits also turn out to be zero.

Algorithm 5 : Yield earlier termination of algorithm 3 for absolutely smaller input. Algorithm 3 is modified exactly as shown.

Input: *integers* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for* $i = 1, \ldots$ *as required in the course of the algorithm; no need for* k
Output: *sign of* x *with very high probability*

2. Terminate the loop also if $y_j = 0$

By lemma 3.1 of [16], this algorithm terminates with a failure with probability bounded by $(k-2)/m_{\min}$, where

$$m_{\min} = \min\{m_1, m_2, \ldots, m_k\}.$$

For $k \leq 12$, $m_{\min} \geq 2^{25}$, the error probability is less than 10^{-6} for uniform distributions. This relies more on the low probability of early termination than on the error probability in case of early termination. But a more careful analysis can reduce the overall probability by exploiting the correlation of failure at different stages and, more importantly, accounting for the non-uniform distribution. For experimental support of this claim, we refer to [16].

6 Applications

Our solutions to problem 1 have many applications. Below we focus on three major areas, namely computation with real algebraic numbers, exact geometric algorithms, and the ubiquitous question of

determinant sign. Additional applications include numeric algorithms for reducing the solution of general systems of analytic equations to sign evaluation [38], deciding the theory of the reals [12, 4], geometric theorem proving [34], and manipulating sums of radicals [2].

6.1 Real algebraic numbers

Being able to compute efficiently with algebraic numbers is important but also necessary in a variety of computer algebra applications, as well as when calculating over the reals. In particular, it is a fundamental operation when computing with algebraic numbers, which is a robust way to treat real numbers, and in general when numeric computation does not offer the required guarantees.

The critical operation is deciding the sign of a multivariate polynomial expression with rational coefficients on a set of points. We will show how our solution can be applied to the manipulation of real algebraic numbers. We refer to [14, 31] for a comprehensive review of the algebraic concepts involved.

A popular paradigm for manipulating algebraic numbers is the use of Sturm sequences. Given two polynomials P and Q in $\mathbb{Z}[X]$, $\deg(P) \geq \deg(Q)$, we consider a *Sturm sequence* $\Sigma = (P_0, P_1, \ldots, P_m)$ of polynomials with $P_0 = P$, $P_1 = Q$, that is, a sequence such that, for all $i = 1, \ldots, m$, $\alpha_i P_{i+1} = \beta_i P_{i-1} + Q_i P_i$ for some integers α_i and β_i and some $Q_i \in \mathbb{Z}[X]$. This implies that $\deg(P_i) < \deg(P_{i-1})$ for all $i = 1, \ldots, m$ and hence that the sequence is finite.

We will assume that P and Q are square-free and do not vanish at a or b. Let $\mathrm{Var}_{P,Q}(a)$ be the number of sign changes of the sequence $\Sigma(a) = (P_0(a), P_1(a), \ldots, P_m(a))$, and define $\mathrm{Var}_{P,Q}[a,b) = \mathrm{Var}_{P,Q}(a) - \mathrm{Var}_{P,Q}(b)$. Sturm sequences have the property that

$$\mathrm{Var}_{P,Q}[a,b) = \sum_{\gamma} \mathrm{sign}(P'(\gamma)Q(\gamma)),$$

where γ ranges over all roots of P in $[a,b)$. Of special interest is the case where Q is the derivative P' of P. In this case, we write $\mathrm{Var}_P[a,b)$ for $\mathrm{Var}_{P,P'}[a,b)$, and this number equals the number of roots of P in $[a,b)$.

It turns out that the coefficients of the P_i's grow very fast, even for simple P and Q. This phenomenon is well known in computer algebra, and seems to require the computations over very large integers. One popular alternative is modular arithmetic. The bottleneck of this approach (at least in theory) is the computation of $\mathrm{Var}_{P,Q}[a,b]$, which involves many sign reconstructions. The recursive relaxation of the moduli is ideally suited because the exact value of $P_i(a)$ is never needed, but only its sign. Therefore, once the sequence Σ is computed in the several finite fields, we may evaluate $\Sigma(a)$ in each finite field and apply algorithm 1 to compute the corresponding sign sequence and finally $\mathrm{Var}_P(a)$.

We examine the complexity of our algorithm for computing the sign sequence corresponding to $\Sigma(a)$ at some rational number a. Let n denote the maximum degree of P and Q, L denote the maximum size of the coefficients of the input polynomials P, Q, and l the sum of the sizes of the numerator and denominator of a. The degrees are decreasing so the length of the sequence is $m \leq n$. As shown in [14], the time to compute the sequence Σ is $O(n^4(L + \log n)^2$, and the coefficient of the P_i's are bounded by $2^{2n(L+\log n)}$. Hence $P_i(a)$ is bounded by

$$|P_i(a)| = n 2^{2n(L+\log n)} 2^{ln},$$

and therefore $O(n(L + l + \log n))$ moduli are sufficient. By using algorithm 1, we correctly retrieve the sign of $P_i(a)$ in time $O(n^2(L + l + \log n)^2)$, for each $i = 0, \ldots, m$. If the sequence is known in each finite field, the computation of the sign sequence corresponding to $\Sigma(a)$ can therefore be done in time $O(n^3(L + l + \log n)^2)$ in the worst case. We summarize this in the following theorem:

Theorem 6.1 *Knowing the Sturm sequence Σ modulo each m_i, $i = 1, \ldots, k$, where $k = O(n(L+l+\log n)$, one can compute $Var_P(a)$ in time $O(n^3(L + l + \log n)^2)$.*

The performance given in the above theorem is in the worst case, however, and in practice, algorithm 1 will run in time $O(k) = O(n(L + l + \log n))$. This lowers the expected complexity of the computation of $\mathrm{Var}_P(a)$ to $O(n^2(L + l + \log n))$ in practice.

As an application of those ideas, we show how to manipulate algebraic numbers. An algebraic number α can be represented symbolically by a square-free polynomial $P \in \mathbb{Z}[X]$ and an interval $I = [a,b]$, such

that α is the only root of P in $[a, b]$ (with multiplicity at least but not necessarily 1). Such an interval can be found by evaluating Var_P at $O(n(L + \log n))$ points [14]. Moreover, in this context, separation bounds imply that $l = O(n(L + \log n))$. The total time of the root isolation procedure is therefore $O(n^6(L + \log n)^3)$. The expected cost is therefore dominated by the sign computations. Practically, however, this cost is expected to be $O(n^4(L + \log n)^2)$, which is the same as the cost of the computation of the Sturm sequence.

To compare two algebraic numbers $\alpha \cong (P, I)$ and $\beta \cong (Q, J)$, we may first assume that they both lie in $I \cap J = [a, b]$, otherwise the comparison can be performed on the intervals. (This assumption can be checked by evaluating Var_P at the endpoints of J and Var_Q at the endpoints of I.) Then (see [14]), $\alpha \geq \beta$ if and only if

$$\mathrm{Var}_{P,Q}[a, b] \cdot (P(a) - P(b)) \cdot (Q(a) - Q(b)) \geq 0.$$

The expensive part of this computation is therefore the computation of $\mathrm{Var}_{P,Q}[a, b]$, which can be done in time $O(n^4(L + \log n)^2$ for the computation of the Sturm sequence and $O(n^3(L + l + \log n)^2)$ for the sign determinations. Practically, the cost of the sign computation is negligible compared to the cost of the computation of the Sturm sequence.

Extension to intersections of algebraic curves can be done in much the same fashion, using multivariate Sturm theory; see [31] and the references thereof. It has been applied in the context of solid modeling by [27] who use modular arithmetic with a *bignum* library for the sign reconstruction.

6.2 Exact geometric predicates

Exact geometric predicates is the most general way to provide robust implementations of geometric algorithms [15, 20, 41, 17]. For instance, orientation and in-circle tests or the comparison of segment intersections, can all be formulated as deciding the sign of a determinant. Before studying the latter question in its own right, we survey several problems in computational geometry which can make use of our algorithms to achieve robustness and efficiency.

Modular arithmetic becomes increasingly interesting when the geometric tests are of higher dimension and complexity. They are central in, notably, convex hull computations: this is a fundamental problem of computational geometry and of optimization for larger dimensions. Computing Voronoi diagram of points reduce to convex hulls in any dimension, but is mostly done in dimensions 2 and 3. Nevertheless, the sweepline algorithm in 2 dimensions involves tests of degree 20 and modular arithmetic can be of substantial help, in conjunction with arithmetic filters [21]. For Voronoi diagrams of segments, the tests become of even higher degree and complexity [11], and f.p. computation is likely to introduce errors, so exact arithmetic is often a must.

Even for small dimensions, the nature of the data may force the f.p. computation to introduce inconsistencies, for instance, in planarity testing in geometric tolerancing [40]. Here, one must determine if a set of points sampling a plane surface can be enclosed in a slab whose width is part of the planarity requirements. The computation usually goes by computing the width of the convex hull, and the data is usually very flat, hence prone to numerical inaccuracies.

In geometric and solid modeling, traditional approaches have employed finite precision floating point arithmetic, based on bounds on the roundoff errors. Although certain basic questions in this domain are now considered closed, there remain some fundamental open problems, including boundary computation [24]. Tolerance techniques and symbolic reasoning have been used, but have been mostly restricted to polyhedral objects; their extension to curved or arbitrary degree sculptured solids would be complicated and expensive. More recently, exact arithmetic has been proposed as a valid alternative for generating boundary representations of sculptured solids, since it guarantees robustness and precision even for degenerate inputs at a reasonable or negligible performance penalty [27]. One key component is the correct manipulation of algebraic numbers (see the previous section).

6.3 Sign of a matrix determinant

Computing the sign of a matrix determinant is a basic operation in computational algebra and geometry, applied to testing the sign of minors, subresultants as well as several geometric tests [31, 5, 13, 3].

To understand the complexity of the problem consider that the entries of the determinant are themselves algebraic expressions. For instance, the in-circle test can be reduced to computing a 2×2 determinant, whose entries have degree 2 and thus require $2b + O(1)$-bit precision to be computed exactly [3]. Computing these entries by using modular arithmetic enables in-circle tests with b-bit precision while still computing exactly the sign of a 2×2 determinant.

To compute an $n \times n$ determinant modulo m_k, we may use Gaussian elimination with a single final division. At step $i < n$ of the algorithm, the matrix is

$$\begin{pmatrix} \vdots & \dots & \dots \\ 0 & \alpha_{i,i} & \dots \\ \vdots & \vdots & \vdots \\ 0 & \alpha_{n,i} & \dots \end{pmatrix}$$

and we assume that the pivot $\alpha_{i,i}$ is invertible modulo m_k. Then we change line L_j to $\alpha_{i,i}L_j - \alpha_{j,i}L_i$ for all $j = i+1, \dots, n$. At step n of the algorithm, we multiply the coefficient $\alpha_{n,n}$ by the modular inverse of the product $\prod_{i=1}^{n-1} \alpha_{i,i}^{n-i}$. This gives us the value of the determinant modulo m_k. Note that the same method but with non-modular integers and a final division would have involved exponentially large integers and several slow divisions at each step. Nevertheless, it is only the range of the final result that matters for modular computations. This shows a big advantage of modular arithmetic over other multiprecision approaches.

The pivots should be invertible modulo m_k. If m_k is prime, the pivot simply has to be non-zero modulo m_k. The algorithm can be also implemented if m_k is a power of a prime, or if m_k is the product of two primes. This would be desirable mainly for taking $m_k = 2^{b_k}$ for which modular arithmetic is done naturally by integer processors, though here, special care must be taken about even output. Other choices of m_k do not seem to bring any improvement.

With IEEE double precision ($b = 53$), we choose moduli smaller than 2^{27}, so that $2(\frac{m_k}{2})^2 \leq 2^{53}$: Gaussian elimination intensively uses $(ad - bc)$-style operations; here we may apply one final modular reduction, instead of two for each product before subtracting.

This algorithm performs $O(n^3)$ operations for each modulus m_i. With Hadamard's determinant bound and m_k greater than $2^{b/2}$, only $k = \lceil 2n \log n \rceil$ finite fields need to be considered. Hence the complexity of finding the sign of the determinant is $O(n^4 \log n)$ single precision operations, when the entries are b-bits integers.

More generally, when the entries are integers of bit-length L, we have to take into account the computation of these n^2 entries modulo m_i, for $i = 1, \dots, k$. Each computation amounts to dividing an L-bit integer by a single-precision integer, in time $O(L)$, for a global cost of $kn^2O(L)$. In this case, Hadamard's bound yields $k = \lceil 2n(\log n + L) \rceil$. Hence, the entire computation takes time $O(n^3(n + L)(\log n + L))$.

To summarize:

Theorem 6.2 *The algorithm described above computes the sign of a $n \times n$ determinant whose entries are integers of bit-length by using $O(n^3(n + L)(\log n + L))$ single precision operations.*

Using the algorithm of Bareiss for this problem yields a bound $O(n^3 M(n(\log n + L))$, where $M(p)$ is the number of operations to compute the product of two p-bit integers. In practice, we almost always have $L = O(n)$. Using multiplication in time $M(p) = p \log p \log \log p$ yields a slightly worse bound than given in the theorem, and with a huge overhead. More practically, using multiplication in time $M(p) = O(p^2)$ yields an order of magnitude slower. Our algorithm is easy to implement and entails little overhead. This is also corroborated by the practical study of section 7.

On a $O(n^3 \log n)$-processor machine, the time complexity drops to $O(n)$, if we use customary parallelization of the Gaussian elimination routine for matrix triangulation (cf. [23]), which gives us the value of the determinant. (We apply this routine in modular arithmetic, with simplified pivoting, concurrently for all m_i's.) Theoretically, substantial additional parallel acceleration can be achieved by using randomization [5, ch. 4], [32], yielding the time bound $O(\log^2 n)$ on $\lceil n^3 \log n \rceil$ arithmetic processors, and the processor bound can be decreased further to $O(n^{2.376})$, by applying asymptotically fast algorithms for matrix multiplication.

To have fewer moduli m_i involved and thus accelerate the computation, we may try to refine the Hadamard bound or to make use of known upper bounds. In particular, such a refinement can be obtained as a by-product of numerical algorithms, which effectively compute the sign of the determinant unless the determinant has a large absolute value [6, 33]. Another way to get a better upper bound is to use a filter with certified arithmetic, such as interval arithmetic. Such a filter will not be able to determine the sign but will return an upper bound which is most of the times much more accurate than Hadamard's bound.

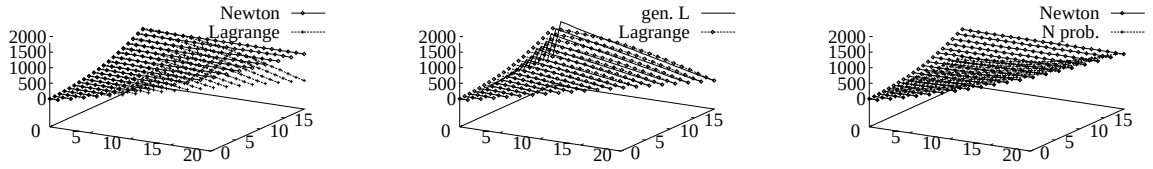
7 Experimental results

7.1 Sign reconstruction in RNS

We present several benchmark results of our diverse methods for reconstructing the sign of an integer $x \in \left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$ represented by its residues $x_i = x \bmod m_k$, $i = 1, \ldots k$. For the data generation, $l \leq k$ residues are chosen at random and the others are computed such that $x \in \left[-\frac{m^{(l)}}{2}, \frac{m^{(l)}}{2}\right)$. The first coordinate is l, the second is k, and the vertical coordinate is the result of the benchmarks, namely the running times of the algorithms.

In figure 7.1(a), we clearly see the k-quadratic behavior of Newton's method for all values of l, and the linear behavior of Lagrange's method near the diagonal $k = l$. We also notice that Lagrange's method is always more powerful than Newton's method. In figure 7.1(b), we see the differences in running times of the standard and the generalized Lagrange methods. They are roughly comparable, except that the generalized method is faster for $l = 0$, because we have implemented the method of our remark 1. This method could have been implemented for all the other methods as well. Finally, we compare the probabilistic and standard Newton's method in figure 7.1(c). We observe that the complexity of the probabilistic method is indeed quadratic in l only, in complementarity with Lagrange's method, which is quadratic in $k - l$. We have not encountered (with our random generation) even a single case of failure for the probabilistic routine.

The measurements are performed on a 200MHz Sun Ultra Sparc workstation. We see for instance that they are negligible with those of the following determinant sign computation, showing that sign determination in RNS using our methods becomes a negligible portion of the determinant sign computation.



(a) (b) (c)

Figure 1: The running times in μs of the different methods, for $0 \leq l \leq k \leq 100$.

7.2 Determinant sign

We present several benchmark results of the described methods for computing the sign of a determinant and compare them with different existing packages. This asserts the practical interest of our algorithms.

- Method FP is a straightforward f.p. implementation of Gaussian elimination which, of course, cannot guarantee correctness of the result. In particular, FP fails for ill-conditioned matrices.

- Method MOD is an implementation of modular Gaussian elimination as described in section 6 using our recursive relaxation of the moduli.
- Method PROB is an implementation of modular Gaussian elimination using the probabilistic Newton variant described in section 5.3, where the computation is stopped when the probability of having a bad result is about 2^{-53}. In all the random matrices we tested, PROB never failed.
- Method CL has been implemented by us based on [13, 10]. As we compare with methods that handle arbitrary dimensions, we did not specialize the implementation for small dimensions as is done in [10] (this would provide an additional speedup of approximately 3).
- Method GMP is an implementation of Gaussian elimination using the GNU Multiprecision Package, for dimension lower than 5, and an implementation of Bareiss' extension of Gaussian elimination, for higher dimensions.
- Method LEDA uses the routine `sign_of_determinant(integer_matrix)` of Leda [11].

All implementations are in C, except LEDA which is in C++. Note that all methods could also be filtered, which would yield running times comparable to those of FP, on random inputs.

To explain the fact that smaller determinants require more time, keep in mind that no special zero test is performed and a static deterministic bound is used on the magnitude. Of the other methods available, the lattice method of [10] has not yet been implemented in dimensions higher than 6; LN [21] provides a very fast implementation in dimensions up to 5 but was not available to us in higher dimensions.

All tests were carried out on a 200MHz Sun Ultra Sparc workstation. Each program is compiled with the compiler that gives best results. Each entry in the following tables represents the average time of one run in microseconds, with a maximum deviation of about 10%. We concentrated on determinant sign evaluation and considered three classes of matrices: random matrices, whose determinant is typically away from zero, in table 1, almost-singular matrices with single-precision determinant in table 2, and lastly singular matrices with zero determinant in table 3. The coefficients are integers of bit-size $53 - n$ (due to restrictions of Clarkson's method).

n	FP	MOD	PROB	CL	GMP	LEDA
2	0.1	7.6	8.0	20.4	6.0	72.0
3	0.8	16.1	17.8	85.5	32.8	296
4	2.0	36.0	39.7	169	136	873
5	3.9	125	123	393	435	2020
6	6.3	186	196	517	1064	4290
7	9.7	329	327	966	2280	8130
8	13.7	472	489	1192	4130	13980
9	19.3	732	747	1894	6540	23100
10	25.8	1021	1038	3080	10810	35100
11	33.8	1400	1423	4240	16200	51600
12	43.4	1865	1898	5530	24200	73900
13	55.5	2610	2640	7570	33000	102200
14	67.3	3330	3370	8580	44500	138100

Table 1: Performance on random determinants.

Among the methods that guarantee exact computation, our implementations are at least as efficient as the others, and for certain classes of input they significantly outperform all available programs. Furthermore, our approach applies to arbitrary dimensions, whereas methods that compute a f.p. approximation of the determinant value are doomed to fail in dimensions higher than 15 because of overflow in the f.p. exponent. The running times are displayed in tables 1–3. (For small dimensions, specialized implementations can provide an additional speedup for all methods.) Our code is reasonably compact and easy to maintain. A possible improvement we plan to explore further is parallelization.

Some side effects may occur, due to the way we generate matrices. The code of the modular package is free, and anyone can benchmark it on the kind of matrices that he uses. It is available via the URL `http://www.inria.fr/prisme/personnel/pion/progs/modular.html`

n	FP	MOD	PROB	CL	GMP	LEDA
2	0.1	8.2	8.0	226	6.4	65.9
3	0.8	18.6	12.3	606	26.8	202
4	2.0	39.8	39.0	1150	85.7	503
5	3.8	121	75.0	2260	174	1078
6	6.3	195	143	3860	524	2020
7	9.7	328	203	6200	852	3680
8	13.9	484	309	8840	2150	5320
9	19.3	745	420	14250	2140	8460
10	26.2	1041	613	19460	3810	11690
11	33.9	1425	789	27000	4270	12960
12	43.3	1899	1199	35900	8300	25000
13	54.6	2630	1365	45000	9030	30600
14	68.1	3370	1986	49200	17270	44200

Table 2: Performance on small determinants.

n	FP	MOD	PROB	CL	GMP	LEDA
2	0.1	9.1	4.7	250	6.4	58.4
3	0.8	19.6	7.4	602	24.1	195
4	2.0	42.5	12.2	1420	91.1	572
5	3.8	124	27.9	2340	216	1330
6	6.3	198	40.0	3980	1095	2920
7	9.7	331	58.0	6170	1751	5640
8	13.8	491	76.4	10900	3940	10140
9	19.3	756	102	13300	5260	22500
10	26.3	1048	133	17400	10560	30600
11	34.4	1432	169	22600	13110	42000
12	43.6	1905	215	34900	24000	68800
13	54.8	2650	265	39400	33100	80300
14	68.0	3380	325	49400	42800	110900

Table 3: Performance on zero determinants.

8 Conclusion

Residue Number Systems (RNS) have been used because they provide a highly parallelizable technique for multiprecision. As parallel and multiprocessor computers are becoming more available, RNS provide an increasingly desirable implementation of multiprecision. This comes in sharp contrast with other multiprecision methods that are not easily parallelizable. Perhaps the main problem with RNS is that comparisons and sign computations seem to require full reconstruction and, therefore, use standard multiprecision arithmetic. We show that one may in fact use only single precision and still perform these operations exactly and efficiently. The speed of the proposed algorithms also relies on their implementation using exclusively f.p. arithmetic.

In some applications, the number of moduli may be large. Our algorithms may be easily implemented in parallel with a speedup depending almost linearly on the number of processors. Another merit of our methods is their simplicity, which makes them attractive to an implementor, and their quasi-linear complexity on the average. Although their worst-case complexity does not achieve the record upper bounds, in practice they appear as the fastest methods today for certain applications.

A relevant application is to compute the sign of a determinant. This problem has received considerable attention in computational geometry, CAD, geometric modeling, as well as symbolic algebra, yet the fastest techniques are usually iterative and do not parallelize easily. Moreover, they usually handle only

single precision inputs. Section 7 shows that our techniques are comparable in speed or even faster than other techniques and can easily handle arbitrarily large inputs.

A central problem we plan to explore further is to design algorithms that compute upper bounds on the quantities involved to determine how many moduli should be taken. For determinants, the static bounds we use seem to suffice for applications in computational geometry [21]. They might be overly pessimistic in other areas (such as tolerancing or symbolic algebra) where the nature of the data or algebraic techniques might imply much better bounds. In this respect, valid approaches include the probabilistic variants introduced above.

As an extension of our algorithms, we may also recover the closest f.p. approximation of a rational number given its modular representation, even though it may not be f.p. representable. This would lead us to a hybrid symbolic/numeric approach to RNS. Such a *filtered RNS* is outlined in [7]. Root isolation as explained in section 6.1 performs this operation for the quite general case of algebraic numbers.

As an application of filtered RNS, there are geometric algorithms whose input is the output of another algorithm. Exact representation of this output would jeopardize the efficient implementations of the subsequent algorithm. A common solution is to round the output of the first algorithm. A key ingredient of these techniques is that every number is rounded to the nearest representable number, so as to ensure that comparisons will not inadvertently be inverted, even though inequalities might become equalities. These renormalization techniques [30, 36] may be implemented using a filtered modular arithmetic and more precisely exact rounding [9].

Acknowledgments

We are thankful to two referees for their suggestions to improve this paper, and to Christoph Burnikel for numerous comments, and for suggesting the linear precomputed tables of section 5.1.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974).

[2] V. Akman and R. Franklin On the Question "Is $\sum_1^n \sqrt{a_i} \leq L$?" Bull. EATCS 28 (1986) 16–20.

[3] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec, Evaluation of a new method to compute signs of determinants, in: Proc. 11th Annu. ACM Sympos. Comput. Geom. (1995) C16–C17.

[4] S. Basu, R. Pollack, and M.-F. Roy, On the combinatorial and algebraic complexity of quantifier elimination, in: Proc. IEEE Symp. Foundations of Comp. Sci. (1994).

[5] D. Bini, V. Y. Pan, Polynomial and Matrix Computations. Vol. 1: Fundamental Algorithms (Birkhäuser, Boston, 1994).

[6] D. Bini, V. Y. Pan, Y. Yu, Certified numerical computation of the sign of matrix determinant, Preliminary report (1997).

[7] J.-C. Bajard, L. S. Didier, J.-M. Muller, A new Euclidean division algorithm for residue number systems, in: Proceedings of ASAP (1996) 45–54.

[8] H. Brönnimann, I. Z. Emiris, V. Pan, and S. Pion, Computing exact geometric predicates using modular arithmetic with single precision, in: *Proc. ACM Symp. on Computational Geometry* (1997) 174–182.

[9] H. Brönnimann and S. Pion, Exact rounding for geometric constructions, in: Proc. of GAMM/IMACS Int. Symp. on Scientific Computing, Comput. Arithm. and Validated Numerics (1997) XIII-1–XIII-3.

[10] H. Brönnimann, M. Yvinec, Efficient exact evaluation of signs of determinants, Research Report 3140 (INRIA Sophia-Antipolis, 1997).

[11] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig, Exact geometric computation in LEDA, in: Proc. 11th Annu. ACM Sympos. Comput. Geom. (1995) C18–C19. Package available at `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[12] J. F. Canny, An improved sign determination algorithm, in: H.F. Mattson, T. Mora, and T.R.N. Rao, ed., Proc. Intern. Symp. Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lecture Notes in Computer Science, Vol. 539 (Springer, Berlin) 108–117.

[13] K. L. Clarkson, Safe and effective determinant evaluation, in: Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci. (1992) 387–395.

[14] G. E. Collins and R. Loos, Real zeros of polynomials, in: B. Buchberger, G.E. Collins, and R. Loos, ed., Computer Algebra: Symbolic and Algebraic Computation (Springer, Wien, 2nd edition, 1982) 83–94.

[15] H. Edelsbrunner and E. P. Mücke, Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms, ACM Trans. Graphics 9:1 (1990) 67–104.

[16] I. Z. Emiris, A complete implementation for computing general dimensional convex hulls, to appear in: Internat. J. Comput. Geom. & Appl. (1997). Preliminary version as Research Report 2551 (INRIA Sophia-Antipolis, France, 1995).

[17] I. Z. Emiris and J. F. Canny, A general approach to removing degeneracies, SIAM J. Computing 24:3 (1995) 650–664.

[18] I. Z. Emiris, V. Y. Pan, and Y. Yu, Modular arithmetic for linear algebra computations in the real field, to appear in: J. Symbolic Computation (1998).

[19] S. Fortune, Numerical stability of algorithms for 2-d Delaunay triangulation, and Voronoi diagrams, in: Proc. 8th Annu. ACM Sympos. Comput. Geom. (1992) 83–92.

[20] S. Fortune and C. J. Van Wyk, Efficient exact arithmetic for computational geometry, in: Proc. 9th Annu. ACM Sympos. Comput. Geom. (1993) 163–172.

[21] S. Fortune and C. J. Van Wyk, Static analysis yields efficient exact integer arithmetic for computational geometry, ACM Trans. on Graphics (1996).

[22] D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Comput. Surv. 32:1 (1991) 5–48.

[23] G. H. Golub and C. F. van Loan, Matrix computations (Johns Hopkins University Press, Baltimore, Maryland, 1996).

[24] C. M. Hoffmann, How solid is solid modeling? in: Applied Computational Geometry, Lecture Notes in Computer Science, Vol. 1148 (Springer, Berlin, 1996) 1–8.

[25] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick, Robust set operations on polyhedral solids, IEEE Comput. Graph. Appl. 9:6 (1989) 50–59.

[26] C. Y. Hung, B. Parhami, An approximate sign detection method for residue numbers and its application to RNS division, Computers Math. Applic. 27:4 (1994) 23–35.

[27] J. Keyser, S. Krishnan, and D. Manocha, Efficient B-rep generation of low degree sculptured solids using exact arithmetic, Technical Report 40 (Dept. Computer Science, Univ. N. Carolina, Chapel Hill, 1996).

[28] D.E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, Vol. 2 (Addison-Wesley, Reading, Massachusetts, 1981 and 1997).

[29] M. Lauer, Computing by homomorphic images, in: B. Buchberger, G.E. Collins, and R. Loos, ed., Computer Algebra: Symbolic and Algebraic Computation (Springer, Wien, 2nd edition, 1982) 139–168.

[30] V. Milenkovic, Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic, in: Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci. (1989) 500–505.

[31] B. Mishra, Algorithmic Algebra (Springer, New York, 1993).

[32] V. Y. Pan, Parallel computation of polynomial GCD and some related parallel computations over abstract fields, Theoret. Comput. Sci. 162:2 (1996) 173–223.

[33] V. Y. Pan, Y. Yu, C. Stewart, Algebraic and numerical techniques for the computation of matrix determinants, Computers & Math. (with applications) 34:1 (1997) 43–70.

[34] A. Rege, A complete and practical algorithm for geometric theorem proving, in: Proc. ACM Symp. on Computational Geometry (1995) 277–286.

[35] N. S. Szabo, R. I. Tanaka, Residue arithmetic and its applications to computer technology (McGraw Hill, 1967).

[36] J.R. Shewchuk, Robust adaptive floating-point geometric predicates, in: Proc. 12th Annu. ACM Sympos. Comput. Geom. (1996) 141–150. .

[37] K. Sugihara and M. Iri, A robust topology-oriented incremental algorithm for Voronoi diagrams, Internat. J. Comput. Geom. Appl. 4 (1994) 179–228.

[38] M.N. Vrahatis, Solving systems of nonlinear equations using the nonzero value of the topological degree, ACM Trans. on Math. Software 14:4 (1998) 312–329.

[39] C. Yap, Towards exact geometric computation, Comput. Geom. Theory Appl. 7 (1997) 3–23.

[40] C. K. Yap, Exact computational geometry and tolerancing metrology, in: D. Avis and J. Bose, ed., Snapshots of Computational and Discrete Geometry, Vol. 3, Tech. Rep. SOCS-94.50 (McGill School of Comp. Sci., 1995).

[41] C. K. Yap and T. Dubhe, The exact computation paradigm, in: D. Du and F. Hwang, ed., Computing in Euclidean Geometry (World Scientific Press, 1995).